The Effect of Java Exceptions on Code Optimisations

Andrew Stevens, Des Watson
School of Cognitive and Computing Sciences
University of Sussex
Falmer, Brighton, BN1 9QH, UK
{andrewst, desw}@cogs.susx.ac.uk

Abstract

The Java language supports exceptions by providing language constructs that allow the programmer to manage any exception that may occur during execution. The Java Virtual Machine (JVM) supports this management by maintaining tables that provide information on the exception handler's location along with the type of exception that it handles. The resulting program's control flow during execution is highly unpredictable and is not amenable to static analysis. Such control flow prediction is a valuable asset to an aggressively optimising compilation system, since, for example, it can identify areas of the program that are likely to be executed most often. Some initial analysis on such programs suggests that the exceptions increase this dynamic behaviour by a factor of four as well as reducing the sizes of basic blocks. Such a result suggests that reducing this effect will increase the potential for compilers to generate faster programs.

1 Introduction

This position paper presents some of the exception-related issues encountered during a research project in the field of compiler-generated code optimisation. The goal of this research is to develop a Java bytecode parallelising compiler that targets an intermediate representation defined by the OSUIF system currently being developed at Stanford University. This paper concentrates on how support for exceptions in the Java language [1] and its Virtual Machine [5] complicates standard compiler analysis techniques, along with proposed techniques that could help to address these problems.

2 Exceptions in Java

There are three levels of support for exceptions in Java programs that are executed on Java Virtual Machines (JVM). The first level is syntactic support in the programming languages via the `try/catch/finally` and `throw` statements. The next level is required in order to represent this support in the compiled bytecode class files. Finally, the runtime JVM implements and manages this support as determined by the bytecodes and the environment that the program executes in. This is a passive exception scheme; the efficiency cost of maintaining the exceptions is nil as long as no exceptions are thrown.

2.1 Exception Categories

Exceptions fit into three different categories: explicit, implicit synchronous and implicit asynchronous. Explicit exceptions occur when a `throw` statement is executed (actually the bytecode `athrow` instruction). These exceptions, defined at the language level, allow the programmer to modify the control flow of the program under definable conditions. Implicit synchronous exceptions, however, are not thrown at locations that the programmer defines. Instead, they are the result of runtime verifications that check that the semantic rules of the language are not broken. These checks include type-checking, array indexing, class loading and resource allocation problems. These are synchronous since they are thrown within a thread at the point that the exception is detected. The implicit, asynchronous exceptions occur for two reasons: an internal error occurs in the JVM or when the `stop` method of class `Thread` is called. These are asynchronous since they can be thrown at any point during the execution of instructions in a thread.

At the language level, all these types of exceptions can be handled using `try` and `catch`. If the type of the thrown exception matches the catch parameter, or is a super-class of the exception class thrown, then the body of the catch is executed. The scope of these handling regions is recorded in an exception table within the class file binary, with one such table per method. Whenever the JVM detects a thrown exception then the table entries (if any) for the current method are examined to find an appropriate handler. If no handler matches the type of exception thrown then the current stack is popped and the exception is re-thrown in the previous frame. This process continues until a handler is located or until all frames for the current thread are popped. If no handlers are found then the `uncaughtException` method is called for the `ThreadGroup` that owns the current thread, which then terminates. The result of this call will generally terminate the program and generate a stack trace. This behaviour can be altered by subclassing `ThreadGroup` and overriding the call.

2.2 Prediction of Control Flow

In order to examine the problems that exceptions create for program analysis by a compiler, it is important to understand the importance of such analysis. Since Java is an interpreted language it has great potential for optimisation. Many systems exist that attempt to translate Java programs into native executables [2, 6, 7, 8]. The more information that a compiler can obtain about a program in advance of execution, the more effect any optimisations will have. If, for example, the compiler can predict the control flow [3], then more aggressive optimisations can be applied to areas most used in execution. Unfortunately, this information is generally not available since control flow can depend on values only available during the execution itself.

Exceptions in Java have a significant effect on such predictions since implicit exceptions are very hard to detect before execution. Certain compiler research projects [4, 6, 2] deal with this problem by not supporting exceptions at all. Another reason for such a decision is the lack of an appropriate representation in a pre-defined intermediate language. Clearly this is not an ideal solution since the behaviour of the program will change and the benefits of exception-based control flow will be lost.

2.3 Related Research

The importance of predicting the dynamics of program execution was examined by Larus [3]. He shows that successful prediction of the control flow leads to the exposure of optimisation opportunities. He also notes the problems caused by the cost of running the tools that identify interprocedural paths. His paper presents another factor that increases the number of dynamic procedural context changes. The introduction of exceptions therefore increases the cost of exposing the executed control flow and therefore decreases the opportunity for optimisations.

The Colorado State University development of a Java compiler called j2s, that uses an intermediate format called SUIF [6], presents compilation issues directly related to the OO qualities of Java. This relates to my research goal of completing an aggressive optimising Java compiler that also makes use of this SUIF format. The main difference is that the j2s system ignores exceptions; this is due to the lack of support within the SUIF representation and related tools. The resulting compiler therefore could not take advantage of any optimisations that apply directly to the SUIF format, in case the program semantics are altered.

Kienle and Hölzle report on the design of a similar compiler, also called j2s, that also targets SUIF [2]. This work also does not currently support exceptions (or threads) but does successfully compile JDK1.2beta2 javac. This j2s system targets a new version of the SUIF representation called OSUIF. The new OSUIF format, along with supporting libraries, contains support for OO languages as well as support for generating exception handling code. The j2s development team plans to extend its support to include exceptions by augmenting the control flow graph to include exception edges.

The instrumentation of Java programs at the class file (bytecode) level has been built into a toolkit by Lee and Zorn [4]. The profiling tool, called "Bytecode Instrumenting Tool", enables a program to insert profiling instructions throughout a Java program to produce information about that program at runtime. Such a system is aimed at providing the system designer with information about the program that may help to point at possible performance bottlenecks. This system also ignores exception support in the language which means that the profiled program may fail a JVM verification check. My research builds on this idea of instrumentation except that my profiled data is intended for compiler usage and not the system designer.

3 Analysis

One of the first stages in any standard control flow analysis is to generate basic blocks. These blocks are areas of code with one entry point and one exit point. The first phase of the research project to parallelise Java programs is the design of an analysis tool that helps measure the effect of exceptions on these basic blocks, and the dynamics of the program in general. The tool uses Java bytecodes as the representation for analysis. This has the key benefit of allowing all external library classes to be analysed as well as to produce results for the whole program.

The results presented in this section were generated by using an analysis tool, written in Java, called jbi. Two Java programs were used to generate results: Sun's javac compiler and the jbi program itself. The first pass of the analysis loads all the class files related to the analysed program into memory. It then steps through each method in each class and generates a control flow graph where each node in the graph is a basic block. Not all methods in a class file can be scanned since some contain no bytecodes. These are either native or abstract methods and are ignored by these tests. The control flow graphs are created using two rules, the first assumes that exceptions can occur and the second assumes that implicit exceptions cannot occur. The Java Virtual Machine specification [5] contains the information for each bytecode instruction, defining any possible exceptions. When exceptions are enabled, an instruction that could potentially generate an exception marks the end of the current basic block. The resulting control flow graphs are stepped through in order to count the block sizes along with the number of possible unpredictable control flow branches. An exception would create an unpredictable branch since the destination handler may not be predicted until runtime.

As can be seen by Table 1, the basic block sizes when exceptions are enabled are significantly smaller when compared to the sizes when disabled. The majority of block sizes for all the tests are in the range of 1–10. A possible reason behind this is shown in Table 2, in the section that counts the number of methods with 10 or fewer instructions. As can be seen, for both programs, the number of methods that fit into this category is about 40% of the total number. Since basic blocks, by definition, cannot cross methods, the blocks in these methods will all be small. The results show that large basic blocks are being broken up into many significantly smaller blocks. Since this break up is due to possible exceptions that a compiler may not be able to rule out, the scope for optimisation is greatly reduced. The effect of these exceptions is to increase the level of control flow dynamics of the program; this can be seen by counting the number of instructions that can generate branches whose destination can only be determined at runtime. An exception would create an unpredictable branch. This count for the jbi program without implicit exceptions is 53208 whereas with exceptions it is 211211. Similarly the same count using the javac program is 27380 without exceptions and 102994 with exceptions. Both these results correspond to around a 400% increase in the program's dynamics with the introduction of exceptions.

		Size of block (number of instructions)								
Program	Exceptions	1–10	11–20	21–30	31–40	41–50	51–60	61–70	71–80	81+
javac	Yes	93231	276	4	0	1	0	0	0	0
	No	54823	1223	128	34	14	5	5	2	32
jbi	Yes	193854	704	30	11	3	3	1	2	0
	No	108657	3758	616	224	65	27	20	13	64

Table 1: Basic Block Size Frequencies

3.1 Exception Prediction

Several possible predictions can be made from examining the different types of exceptions. The potential for explicit exceptions is easily recognised by the presence of the `athrow` bytecode. However, the destination of the handler for such exceptions will depend on the exception type, which is a runtime value. In the current

	No. of classes	Methods		
		Total Number	Implemented	Size 1 to 10
javac	740	7226	6583	3023
jbi	1984	18309	15998	7495

Table 2: Program Statistics

phase of the research project, the bytecode class files relating to a program are being instrumented with bytecodes that will allow runtime information regarding these exceptions to be generated. This information could help to obtain the destination of explicit exceptions and prove that certain implicit exceptions are not possible.

3.2 Current Work

At present, the prototype bytecode instrumenter conducts some basic static analysis on a Java program before instrumenting the classfiles. The instrumented program can then be run to extract information about the control flow and execution time. The next step will be to insert the instrumentation code that will provide information on potential exceptions during the execution. This information can then be fed back into the compiler to assist with reducing the effects of such exceptions.

4 Conclusions

Exceptions in Java add many benefits to the design and implementation of a program from enforcing caller handling of errors to recovering from system resource limits. As a strongly typed OO language, Java adds to these benefits by enabling the compiler to validate that explicit exceptions are being handled in the surrounding scope. With these benefits come disadvantages. The fragmentation of basic blocks within the control flow has a negative impact on the types of optimisation that can be applied. The effect of these exceptions also increases the dynamics of the program's control flow, reducing the predictive powers of an ahead-of-time compiler. A compilation system that can profile the runtime code in an attempt to reduce the number of possible runtime exceptions could help to reduce this impact. A solution based on profiling in advance of compilation is not ideal since it is potentially time consuming. A possible solution would be to integrate the profiler into a Just In Time (JIT) compiler system or one that provides runtime feedback for subsequent compilations.

References

[1] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996. `http://java.sun.com/docs/books/jls/index.html`.

[2] Holger Kienle and Urs Hölzle. j2s: A SUIF Java compiler. Technical report, Department of Computer Science, University of California Santa Barbara, August 1997. Also available as Technical Report TRCS97-16.

[3] James Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, 1999.

[4] Han Bok Lee and Benjamin G. Zorn. Bit: A tool for instrumenting Java bytecodes. In *USENIX Symposium on Internet Technologies and Systems Proceedings*, pages 72–83, Monterey, California, 1997.

[5] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, September 1996. `http://java.sun.com/docs/books/vmspec`.

[6] Sumith Mathew, Eric Dahlman, and Sandeep Gupta. Compiling Java to SUIF: Incorporating support for object-oriented languages. In *Proceedings of the Second SUIF Compiler Workshop*, August 1997.

[7] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Berkeley, June 16–20 1997. Usenix Association.

[8] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications: A way ahead of time (WAT) compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 41–54, Berkeley, June 16–20 1997. Usenix Association.